VersionClimber: version upgrades without tears

Christophe Pradal, Sarah Cohen-Boulakia, Patrick Valduriez, Dennis Shasha

VersionClimber: version upgrades without tears

Christophe Pradal UMR AGAP, CIRAD and Inria, Univ. Montpellier, Montpellier, France
christophe.pradal@cirad.fr
Sarah Cohen-Boulakia LRI CNRS 8623, U.Paris Sud, France
cohen@lri.fr
Patrick Valduriez Inria and LIRMM, Univ. Montpellier, France
patrick.valduriez@inria.fr
Dennis Shasha Courant Institute, New York University, USA
Inria and LIRMM, Montpellier, France
shasha@cs.nyu.edu

Abstract

VersionClimber is an automated system to help update the package and data infrastructure of a software application based on priorities that the user has indicated (e.g. I care more about having a recent version of this package than that one). The system does a systematic and heuristically efficient exploration (using bounded upward compatibility) of a version search space in a sandbox environment (Virtual Env or conda env), finally delivering a lexicographically maximum configuration based on the user-specified priority order. It works for Linux and Mac OS on the cloud.

Index Terms

Software Engineering, Versioning, conda, software deployment,

I. Introduction

Data scientists, natural scientists, and enterprise-scale application programmers typically need to analyze multiple data sources using multiple packages, each of which evolves independently. As professionals primarily concerned with their domains, the data scientists have neither the interest nor the technical skill to engage in the continuous development protocols [1], [2] that are currently considered the best practice to keep up with recent versions of each package. More likely, they find themselves needing to update the packages or data sources of some software system (often implemented as a workflow) whose original developer has left or whose project was put on hold.

Because updating the version of one package may break compatibility with other packages, one approach to explore this package-version space is manual testing among version combinations, but that proves to be frustratingly time-consuming given the number of combinations to be tested.

In the current state of the art, a data scientist might use conda as a multi-OS package manager because of its ability to build conda environments that can contain different versions of different packages. A conda environment is isolated from the operating system and can be exported as a set of versioned packages to be cloned on another computer. However, updating an environment often produces errors due to conflicts between different versioned packages, so updating a set of packages to later versions remains a trial-and-error process when using conda.

A complementary approach to support isolation is to use containers, such as Docker containers, to install a set of package-versions on a given Operating System in an image that can be stored on the web and deployed on diverse computer and operating systems. While this supports both isolation and reproducibility across platforms, building a docker image requires substantial engineering knowledge and the frozen image cannot evolve through time. Evolution remains manual.

An automatic approach to updating versions is an active area of research in the Linux kernel community (e.g. https://kernelnewbies.org/KernelProjects/kconfig-sat). The idea is to encode the dependencies between package-version pairs and then to use a satisfiability (SAT) solver to find compatible versions of package-versions [3]. This could work well for the Linux kernel where the number of packages is limited and well known, but knowing these compatibility relationships in a system put together in a one-of-a-kind way is infeasible in general.

VersionClimber is an automated system to help update the package and data infrastructure of a software application based on package priorities that the user has indicated. Based on configuration files and a notion of success and failure that the user sets up, VersionClimber does an efficient exploration through version space finally arriving at a lexicographically maximum configuration based on that priority order.

We will show two examples to illustrate VersionClimber in action. The software (source and binary) is available at the addresses given in the summary.

II. Laser Interferometer Gravitational-Wave Observatory (LIGO) Case Study

Our first example is a data science pipeline used in physics and developed in Python to analyse LIGO data. A reproducible pipeline built on a Jupyter Notebook exists to analyze gravitational waves data. The notebook uses the following packages: Python, NumPy, SciPy, Jupyter Notebook, H5Py and also plotting libraries such as Matplotlib and SeaBorn. While all these libraries are quite standard, the Notebook depends on Python 2 and fails with Python 3.

Let us say that another physics group wanted to update those packages to their latest versions to use them for a new application of gravitational interferometry. VersionClimber allows such a group to specify the package order in descending order of priority (of which ones they most want to bring up to date) and the versions of each package to be considered and finds the most up-to-date configuration that works based on the versions the user would like to consider for each package. We now present a step by step tutorial for how to do this.

A. Updating the LIGO software using VersionClimber

First, install conda: create a conda environment *cise* with VersionClimber installed in it, and activate it:

```
conda create --name cise -c versionclimber versionclimber -y
conda activate cise
```

Then, to reproduce the LIGO analysis pipeline, create a directory containing a configuration file named *config.yaml* available at https://github.com/VersionClimber/VersionClimber/blob/master/example/tuto_ligo/config.yaml. VersionClimber uses this declarative configuration file to indicate which packages have to be tested and their priority order.

```
pre: git clone https://github.com/pradal/ligo-binder.git ligo
packages:
    - name      : scipy
      vcs       : conda
      cmd       : conda install -y
      hierarchy : patch
      supply    : minor

    - name      : numpy
      vcs       : conda
      cmd       : conda install -y
      hierarchy : patch
      supply    : minor

    - name      : seaborn
      vcs       : conda
      cmd       : conda install -y
      hierarchy : patch
      supply    : minor

    - name      : h5py
      vcs       : conda
      cmd       : conda install -y
      hierarchy : patch
      supply    : minor

    - name      : notebook
      vcs       : conda
      cmd       : conda install -y
      hierarchy : patch
      supply    : minor

    - name      : nbconvert
      vcs       : conda
      cmd       : conda install -y
      hierarchy : patch
      supply    : minor

    - name      : matplotlib
      vcs       : conda
      cmd       : conda install -y
      hierarchy : patch
      supply    : patch

    - name      : python
      vcs       : conda
      cmd       : conda install -y
      hierarchy : patch
      supply    : major


run:
    - jupyter nbconvert --ExecutePreprocessor.timeout=60 --to notebook
    --execute ligo/index.ipynb
```

```
post: conda env export > environment.yml
```

The YAML file is divided into four sections, namely **pre**, **packages**, **run**, and **post**:

- **pre (Optional):** A set of commands to execute before running VersionClimber. Here, we will clone from github the repository ligo-binder that contains a notebook *index.ipynb* which will be run on each potential configuration of package-versions.
- **packages:** A package is declared using a set of meta-information such as:
  - **name** is the name of the package.
  - **vcs** is the type of version control system the package use (i.e. git or svn for source packages or pypi and conda for binary packages).
  - **url** is the address where the package will be cloned or checkout. (This information is needed only for source packages.)
  - **cmd** is the command to build the package.
  - **conda** is an optional argument to indicate if the package is managed by conda (True) or pip (False).VersionClimber supports only these two package managers because they support package installation across multiple platforms in user space (i.e., without administrator privileges).
  - **recipe** is the local path where the conda recipe is defined. This optional parameter is defined when building executables from source.
  - **channels** is a list of priority channels to consider when installing with conda.
  - **hierarchy** is a way to describe the search space for a package, i.e. the set of versions to consider. The possible values are *major*, *minor*, *patch* and *commits* (default). For example, in the case of "hierarchy:patch", VersionClimber will consider the set of versions consisting of the last commit of each patch. By contrast "hierarchy:minor" means that VersionClimber will consider only the latest commit of the latest patch of each minor version. The larger the search space the more work, but potentially also the slightly more up-to-date configuration that will come out.
  - **supply** indicates a list of consecutive versions of a package that are backward compatible in the sense that every version in the list supplies the same functions with the same signatures (i.e. same API). So, for example, saying "supply: minor" means that all versions of a package that share the same major and minor version designations provide exactly the same functions with the same signatures and the same semantics. They are said to be "supply-constant." When semantic versioning doesn't apply, then this will require user knowledge. If that knowledge is not available, then the user can set this parameter to demand: X where X is the same value as given as a parameter to hierarchy. That will slow things down (perhaps by a lot) but may result in a more recent version of this package.
    The "anchor" of a supply-constant sequence of versions is the first member of the sequence, because that version provides the same services as the others and demands the least.
    demand: By contrast, the line "demand:X" (e.g., demand: major) means that all versions that have the same X (in this example, major) designation are backward compatible and all those versions require (i.e., "demand") the same function-signatures from other packages. They are said to be "demand-constant". The "anchor" of a demand-constant sequence of versions is the last member of the sequence, because that version provides the most services of any in the sequence without demanding more. We formalize these notions in the next subsection.
- **run:** the command to test the different packages together to know if a configuration is valid. In this example, the Jupyter Notebook is executed and its return status indicate if the packages at a given version are compatible (success) or not (failure). A user may include other commands that execute a test suite or a script.
- **post (Optional):** a command run at the end of the process, when VersionClimber has found the up-to-date package combination. Here, we export the conda environment into a file to be able to recreate it on another computer. Other options are possible, e.g., build a Docker image.

To run VersionClimber in the directory containing the *config.yaml* file, just use the command:

```
vclimb -a
```

The number of potential combinations for the LIGO application is very large: 53,212,287,744. The number of anchors (see next subsection) to explore is much smaller, but remains quite large: 45,208,800. The versions of each package and the number of configuration can be obtained by running the following command:

```
vclimb --version
```

The optimal working configuration is found after testing 9 configurations in 1050s. (Working in this case means simply "not crashing", but in our next example, we show how to integrate user-written tests to indicate what working means.)

```
---
Configuration 1
scipy: 1.2.0, numpy: 1.16.0, h5py: 2.9.0, seaborn: 0.9.0, notebook: 5.7.0,
nbconvert: 5.4.0, matplotlib: 3.0.2, python: 3.7.3
```

```
FAILURE
---
Configuration 9
scipy: 1.2.1, numpy: 1.16.3, h5py: 2.9.0, seaborn: 0.9.0, notebook: 5.7.8,
nbconvert: 5.4.1, matplotlib: 2.2.3, python: 2.7.16

Run conda install -y  scipy=1.2.1 numpy=1.16.3 h5py=2.9.0 seaborn=0.9.0 notebook=5.7.8
nbconvert=5.4.1 matplotlib=2.2.3 python=2.7.16 in 26.595161 s

jupyter nbconvert --ExecutePreprocessor.timeout=60 --to notebook --execute ligo/index.ipynb
Configuration execution in 15.624013 s
SUCCESS
---

Total time in 1050.414971 s
```

B. Supply-constant and Demand-constant – the building blocks of VersionClimber

Definition: The *supply set* of a package-version P.v, denoted supplyset(P.v) is the set of function-signatures that P.v supports.
Definition: A consecutive series of versions P.vbottom ... P.vtop is *supply monotonic* (commonly known as backward compatible) if the supplyset(P.v1) is a subset of supplyset(P.v2) for all v1, v2 such that vbottom $\leq$ v1 $\leq$ v2 $\leq$ vtop. That is, the supply set is monotonically increasing and every function preserves its semantics once it is included in some version.
Definition: The *demand set* of P.v, denoted demandset(P.v), are the external functions called by P.v in the application. Each function is uniquely defined by the package it comes from and the types of its arguments and the type of its return value.
Definition: A consecutive series of versions are *demand monotonic* if the demandset(P.v1) is a subset of demandset(P.v2) for all v1, v2 such that vbottom $\leq$ v1 $\leq$ v2 $\leq$ vtop. That is, the demand set is monotonically increasing.
Definition: A *well-formed mini-series* is supply monotonic and demand monotonic.
Definition: A *demand-constant mini-series* Pi.vbottom ... Pi.vtop is well-formed and has the additional property that demandset(Pi.vtop) = demandset(Pi.vbotom). That is, the demand set doesn't change.
Definition: A *supply-constant mini-series* Pi.vbottom ... Pi.vtop is well-formed and has the additional property that supplyset(P.vbottom) = supplyset(P.vtop).

Note that in a semantic versioning setting, patch versions within the same major-minor release are often supply-constant. Also note that a single version is always demand-constant and supply-constant. Pragmatically speaking, that means the user can always choose to try every version of a package.

III. Multi-Lingual Case study: Phenomenal

Our second example uses packages from multiple languages (*i.e.* C++ and Python) to process biological image data and extract three-dimensional structural information to infer genotype to phenotype relationship.

A Python library, named Phenomenal, has been developed and depends on a set of widely used libraries for scientific visualisation and image processing such as VTK and OpenCV. While these libraries have Python interfaces, they are implemented in C++ for efficiency. However, they are known to be difficult to install due to binary conflicts and pairwise dependencies.

In addition, Phenomenal relies on some code that is not available on any public conda channel. For each configuration, we will rebuild the local packages. Then, to test the status of each configuraion, rather than executing a notebook, we will run the full test suite. That is what "working" will mean in this case.

Phenomenal depends on Python packages such as Python, NumPy, SciPy, MatPlotLib, NetworkX, Pandas, Scikit-Learn, Scikit-Image, and OpenCV and VTK. It depends also on utilities such as nose, coverage or openalea.deploy.

The complete example is available at https://github.com/VersionClimber/CiSE_multilanguage.

The user might create the *Phenomenal* configuration as follows:

```
packages:
    - name       : openalea.deploy
      vcs        : git
      url        : https://github.com/openalea/deploy.git
      build_cmd  : conda build
      cmd        : conda install -y --use-local
      conda      : True
      recipe     : recipes/deploy
      hierarchy  : patch
```

```
  supply     : minor

- name       : openalea.phenomenal
  vcs        : git
  url        : https://github.com/openalea/phenomenal.git
  build_cmd  : conda build
  cmd        : conda install -y --use-local
  conda      : True
  recipe     : recipes/phenomenal
  hierarchy  : patch
  supply     : minor

- name       : nose
  vcs        : conda
  cmd        : conda install -y
  hierarchy  : patch
  supply     : minor

- name       : coverage
  vcs        : conda
  cmd        : conda install -y
  hierarchy  : patch
  supply     : minor

- name       : pandas
  vcs        : conda
  cmd        : conda install -y
  hierarchy  : patch
  supply     : minor

- name       : vtk
  vcs        : conda
  cmd        : conda install -y
  hierarchy  : patch
  supply     : minor

- name       : opencv
  vcs        : conda
  cmd        : conda install -y
  hierarchy  : patch
  supply     : minor

- name       : networkx
  vcs        : conda
  cmd        : conda install -y
  hierarchy  : patch
  supply     : minor

- name       : scikit-image
  vcs        : conda
  cmd        : conda install -y
  hierarchy  : patch
  supply     : minor

- name       : scikit-learn
  vcs        : conda
  cmd        : conda install -y
  hierarchy  : patch
```

```
    supply     : minor

  - name       : scipy
    vcs        : conda
    cmd        : conda install -y
    hierarchy  : patch
    supply     : minor

  - name       : cython
    vcs        : conda
    cmd        : conda install -y
    hierarchy  : patch
    supply     : minor

  - name       : numba
    vcs        : conda
    cmd        : conda install -y
    hierarchy  : patch
    supply     : minor

  - name       : numpy
    vcs        : conda
    cmd        : conda install -y
    hierarchy  : patch
    supply     : minor

  - name       : matplotlib
    vcs        : conda
    cmd        : conda install -y
    hierarchy  : patch
    supply     : patch

  - name       : python
    vcs        : conda
    cmd        : conda install -y
    hierarchy  : patch
    demand     : major

run:
  - nosetests -w .vclimb/openalea.phenomenal/test

post: conda env export > environment.yml
```

All the packages are installed from conda using the *default* channel. Only two packages, namely openalea.deploy and openalea.phenomenal are built from source. The source code is retrieved from github. For each tagged version, a conda package is built then installed with the other dependencies.

Because this example requires re-compilation, the user must create a conda "recipe". The recipe is the part of the requirements following "build-cmd:" and lists the names of the packages under test. The particular package-versions are taken from those installed from conda. In the example below, pandas version 0.23.4, numba 0.41.0 etc.

An example of an openalea.phenomenal recipe is the following. The $version notation causes VersionClimber to specify the version of the openalea.phenomenal package to conda. One should use this for any package that is recompiled for each configuration as we describe in more detail in the documentation.

```
package:
  name: openalea.phenomenal
  version: "$version"

source:
  path: ../../.vclimb/openalea.phenomenal
```

```
build:
  number: 0
  preserve_egg_dir: True
  string: phenomenal
  script:
    - python setup.py install --single-version-externally-managed --record record.txt

requirements:
  build:
    - python
    - setuptools
    - openalea.deploy
    - cython
    - numpy

  run:
    - python
    - numpy
    - numba
    - cython
    - openalea.deploy
    - scipy
    - scikit-image
    - scikit-learn
    - networkx
    - opencv
    - matplotlib
    - vtk
    - pywin32 [win]
    - nose
    - coverage
    - sphinx
```

There are a total of 28,669,893,502,228,070,400 possible configurations. Alternatively, a simple heuristic to find the optimal configuration is to test the configurations in a lexicographically descending order, based on the priorities of the packages. That would require testing 5963 configurations requiring over 10 days. We stopped the process after testing 730 configurations over 30 hours. VersionClimber by contrast found the optimal solution in 8 hours and 48 minutes after testing 78 configurations.

The result is

```
openalea.deploy =v2.1.1
openalea.phenomenal =v1.6.1
nose =1.3.7
coverage =4.5.3
pandas =0.24.2
vtk =8.2.0
opencv =4.1.0
networkx =2.2
scikit-image =0.14.2
scikit-learn =0.20.3
scipy =1.2.1
cython =0.29.7
numba =0.43.1
numpy =1.15.4
matplotlib =2.2.3
python =2.7.16
```

IV. Summary

VersionClimber helps developers upgrade their systems by choosing configurations efficiently and then testing them automatically. These configurations may consist of source packages as well as binary ones and may be implemented using multiple languages. VersionClimber has been tested on both Linux and Mac OSX, though the software is also deployable on Windows. VersionClimber uses the package managers pip and conda, because of their support for multiple platforms and their ability to load packages into user space.

VersionClimber is open source and is available at
https://github.com/VersionClimber/VersionClimber with the documentation available here:
https://versionclimber.readthedocs.io

References

[1] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.
[2] C. Larman, *Agile and iterative development: a manager's guide*. Addison-Wesley Professional, 2004.
[3] P. Trezentos, I. Lynce, and A. L. Oliveira, "Apt-pbo: solving the software dependency problem using pseudo-boolean optimization," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 427–436.